

CMSC 433 – Programming Language Technologies and Paradigms Spring 2003

Java Language Runtime
May 6, 2003

Project 6

- Hack: Can set codebase with
 - `System.setProperty("java.rmi.server.codebase", s);`
 - Do this before any RMI calls
 - (Probably can't use `setProperty` for `java.policy`)
- Multicast code available soon
 - Should incorporate into your project
- RMI check tool available soon
 - Will let you see how an object is registered

2

Portable Code

- Most languages compiled to “native” code
 - E.g., `gcc` produces `x86/ppc/sparc/...` object code
 - Object code in CPU's instruction set
 - This is what goes into executable (`windows .exe`)
 - Object code is platform-specific
- Programs are also OS specific
 - Can't do much without using API for OS

3

Java Byte Code

- Java compiled into JVM byte code (class files)
 - In instruction set for *Java Virtual Machine*
 - Not directly executable by any real CPU
 - Instead, either interpreted or compiled to native code
 - Java runtime takes place of OS APIs
 - E.g., calls to `System.out.println` eventually go to OS
 - Name that design pattern
- Goal: “Write once, run anywhere”
 - Is this really true?

4

Goals of Java Virtual Machine

- Simple instruction set
 - E.g., not `x86`
- Small files
 - Easy to send over the internet
- Portable
 - Easy to interpret/compile on many different architectures

5

Data Types

- JVM has same data types as Java
 - `int`, `char`, `double`, etc.
 - objects and arrays
- No boolean type in JVM
 - Uses `int` instead
- JVM data stored in words
 - One word big enough for byte...float, object ref
 - Two words big enough for long, double

6

JVM Architecture

- The JVM is a *stack* machine
 - (Almost) all instructions operate on the stack
 - Contrast to a register machine (e.g., standard CPU)

```
iconst_2    // push 2 on the stack
iconst_3    // push 3 on the stack
iadd        // add top two stack elements
```

- Stack contains sequence of words
 - Larger data (e.g., long, double) takes 2 words

7

Constant Pool

- Each class file contains a *constant pool*
 - Stores strings, big integers, longs, names and signatures of methods, etc.

- Can push constants onto the stack

```
ldc #2      // push 35423 onto stack
iconst_1    // push 1 onto stack
iadd        // add top two stack elements
```

8

Method Sigs in Constant Pool

- Stored as a string:
 - Object mymethod(int i, double d, Thread t);
 - (IDLjava/lang/Thread;)Ljava/lang/Object;
 - (Example from JVM Spec book)
- Gets in the way of small class files!
 - That's why jar files are compressed

9

Local Variables

- Can also load/store from local variables

```
iconst_3    // push 3 on the stack
istore_1    // store in local variable 1
```
- Local variable #0 always set to *this*
- Multi-word data takes two local variables

```
ldc2_w #2   // push (long) 3 on stack
istore_1    // store in local variables 1 and 2
```

10

Method Calls

- [this.]foo(1, "Hello")

```
aload_0     // push this on stack
iconst_1    // push 1 on stack
ldc #4      // push "Hello" on stack
invokevirtual #5 // call foo(int, String)
```

– If foo() has result, pushed on stack after invokevirtual

11

Method Calls (cont'd)

- Actual parameters stored in local variables
 - E.g., foo()'s int param stored in local var 1

```
iload_1     // push param on stack
```
- Method ends with **Xreturn** instruction
 - X = a (object), l (long), i (int), etc.
- Conceptually, each method has own stack, local variables
 - No way to access stack frame or local variables of any other method

12

Program Counter and Exceptions

- Methods contain numbered seq of instructions
 - Every instruction is one byte wide (202 total)
 - Followed by operands

```
0 iload_1
1 ldc #2 <Integer 345423>
3 iadd
4 istore_3
```

- Each method has an exception table
 - Maps instructions to exception type, target

from	to	target type
0	5	8 <Class java.lang.NullPointerException>

Java Verifier

- Checks
 - Class file has correct format
 - Methods called with correct #, types of args
 - Looks at types of data on stack at time of invoke
 - Stack has same shape at joins
 - I.e., two jumps to same location
 - Methods return data of correct type
 - Stack size of method is bounded
 - etc.

14

Use javap to View Class Files

- javap -c MyClass

```
...
Method void main(java.lang.String[])
0 getstatic #2 <Field java.io.PrintStream out>
3 ldc #3 <String "Hello, world.">
5 invokevirtual #4 <Method void
  println(java.lang.String)>
8 return
```

15

Class Loaders

- Java classes and interfaces can come from many places
 - Local .class file
 - .class file downloaded from network
 - Proxy reflection classes
 - ...
- Class/interface loaded into memory when
 - Referenced (need to check types)
 - Gotten via another means (e.g., RMI)

16

Class Loaders (cont'd)

- Can implement your own class loader
 - RMI class loader
 - checkSync has a class loader that rewrites bytecode to keep track of lock/unlock
- Internally, class names in JVM combine both name and class loader
 - That way no one can replace your local java.lang.String class

17

Just-in-Time Compilation

- Java byte codes can't run directly on CPU
 - Need to interpret them
 - Which slows things down
- Or, can compile JVM code to native code
 - Compile code when loaded by class loader
 - Can't compile earlier, since new classes can be created dynamically

18

Memory Management in Java

- Local variables live on the stack
 - Allocated at method invocation time
 - Deallocated when method returns
- Other data lives on the heap
 - Memory is allocated with new
 - But never explicitly deallocated
 - Java uses automatic memory management

19

Garbage Collection (GC)

- At any point during execution, can divide the objects in the heap into two classes:
 - Live objects will be used later
 - Dead objects will never be used again
 - They are garbage
- Idea: Can reuse memory from dead objects

20

GC Techniques and the JVM

- The JVM Specification doesn't say how to manage the heap
- Simplest valid memory management strategy: never delete any objects
 - Not such a bad idea in some circumstances (when?)

21

Many GC Techniques

- We can't know for sure which objects are really live or dead
 - Undecidable, like solving the halting problem
- Thus we need to make an approximation
 - OK if we decide something is live when it's not
 - But we'd better not deallocate an object that will be used later on

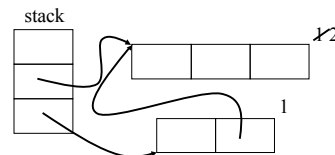
22

Reference Counting

- Old technique (1960)
- Each object has count of number of pointers to it from other objects and from stack
 - When count reaches 0, object can be deallocated

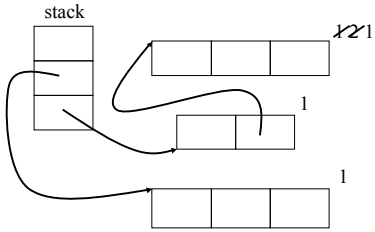
23

Reference Counting Example



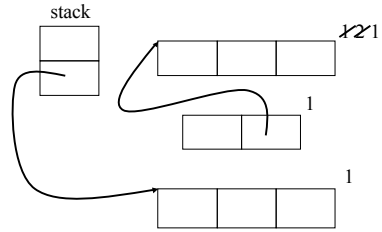
24

Reference Counting Example



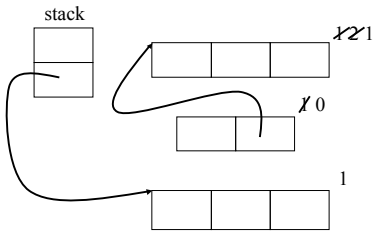
25

Reference Counting Example



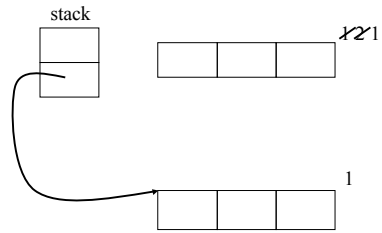
26

Reference Counting Example



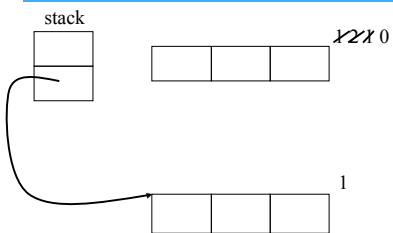
27

Reference Counting Example



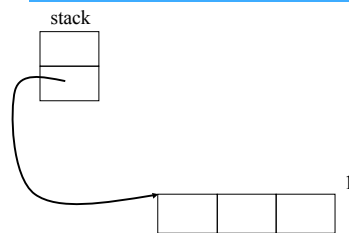
28

Reference Counting Example



29

Reference Counting Example



30

Tradeoffs with Ref Counting

- Advantage: Incremental technique
 - Small amount of work per memory write
 - With more effort, can bound running time
 - Useful for real-time systems
- Problem: Data on cycles can't be collected
 - Counts never go to zero

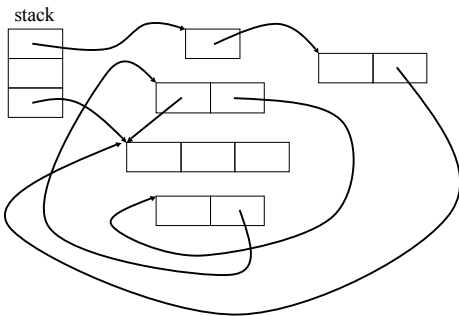
31

Mark and Sweep GC

- Idea: Only objects reachable from stack could possibly be live
 - Every so often, stop the world and do GC:
 - Mark all objects on stack as live
 - Until no more reachable objects,
 - Mark object reachable from live object as live
 - Deallocate any non-reachable objects
- This is a *tracing* garbage collector

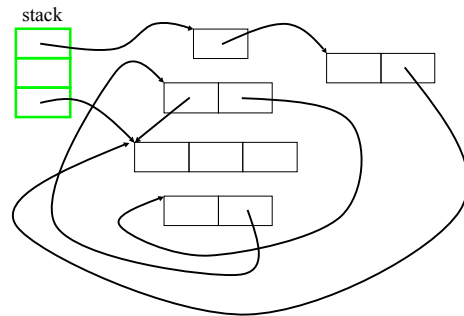
32

Mark and Sweep Example



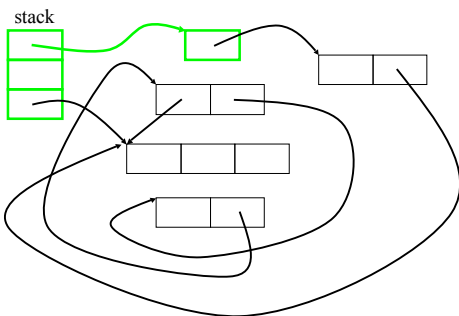
33

Mark and Sweep Example



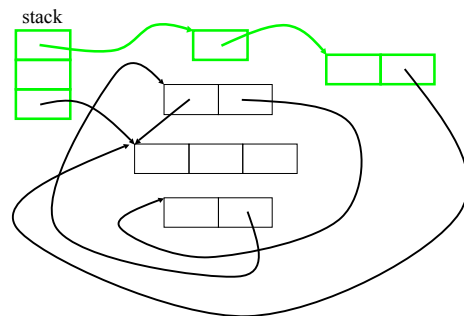
34

Mark and Sweep Example



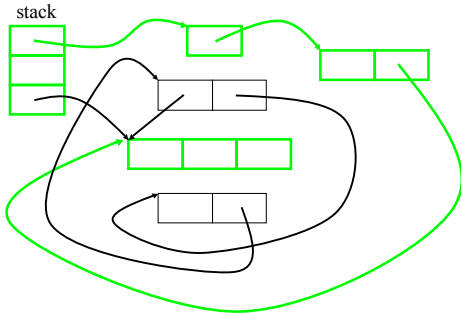
35

Mark and Sweep Example



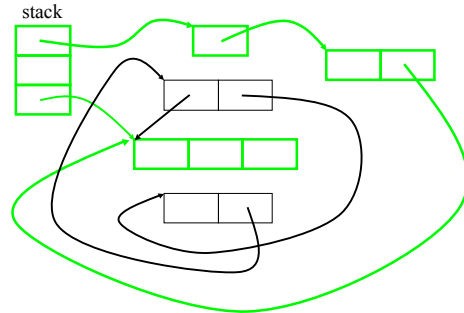
36

Mark and Sweep Example



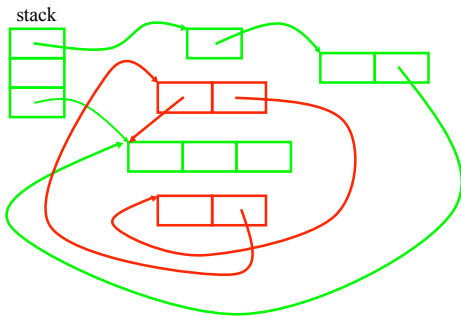
37

Mark and Sweep Example



38

Mark and Sweep Example



39

Tradeoffs with Mark and Sweep

- Pros:
 - No problem with cycles
 - Memory writes have no cost
- Cons:
 - Fragmentation
 - Cost proportional to heap size
 - Sweep phase needs to traverse whole heap

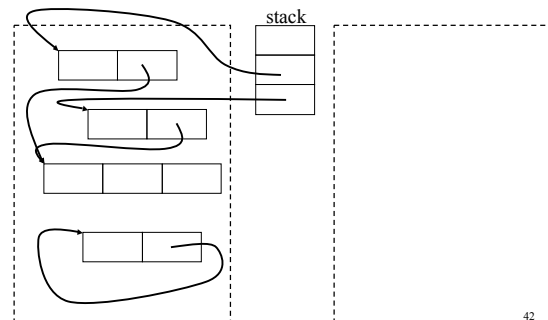
40

Stop and Copy GC

- Like mark and sweep, but only touches live objects
 - Divide heap into two equal parts (semispaces)
 - Only one semispace active at a time
 - At GC time, flip semispaces
 - Trace the live data starting from the stack
 - Copy live data into other semispace
 - Declare everything in current semispace dead; switch to other semispace

41

Stop and Copy Example



42

